13015 计算机系统原理

第四章考点解析（1）

13015 计算机系统原理【回顾第一章内容】

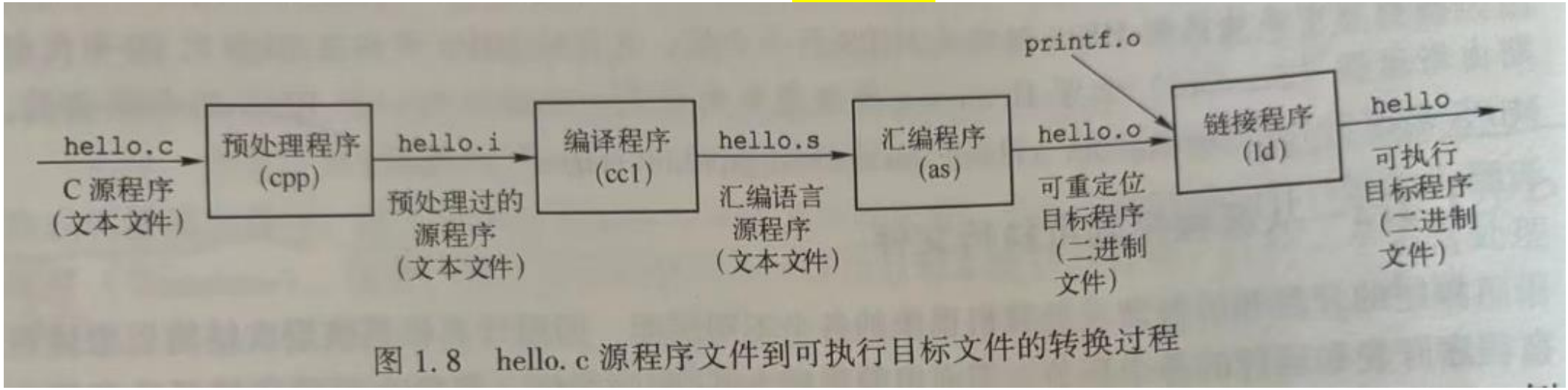


图 1.8　hello.c 源程序文件到可执行目标文件的转换过程

1、源程序文件到可执行目标文件的转换过程是什么？**简答题，填空题**

1）**预处理阶段**：预处理程序(cpp)对源程序中以字符“#”开头的命令进行处理，以**.i**为扩展名。-->**预处理过源程序，文本文件**

2）**编译阶段**：编译程序(ccl)对预处理后的源程序进行编译，生成一个汇编语言源程序文件，以**.s**为扩展名。-->**汇编语言源程序，文本文件**

3）**汇编阶段**：汇编程序(as)对汇编语言源程序进行汇编，生成一个**可重定位目标文件**，以**.o**为扩展名。-->**二进制文件，打开是乱码**

4）**链接阶段**：链接程序(ld)将多个可重定位目标文件和标准库函数库中的可重定位目标文件合并成为一个**可执行目标文件**。-->**二进制文件，打开是乱码**

13015 计算机系统原理【第四章】

考点1 可执行文件生成

1）预处理阶段

预处理命令是：**gcc -E** 或 **cpp**

例如：可用命令“**gcc -E** main.c -o main.i”或“**cpp** main.c -o main.i”

将main.c转换为预处理后的文件main.i

2）编译阶段

预处理命令是：**gcc -S** 或 **ccl**

例如：可用命令“**gcc -S** main.i -o main.s”或“**ccl** main.i -o main.s”

对main.i进行编译并生成汇编代码文件main.s

13015 计算机系统原理【第四章】

考点1 可执行文件生成

3）汇编阶段（汇编语言代码→机器语言代码）

汇编命令是：**gcc -c** 或 **as**

例如：可用命令“**gcc -c** main.s -o main.o”或“**as** main.s -o main.o”

将main.s进行汇编，以生成**可重定位文件**main.o

4）链接阶段

链接命令是：**gcc -o** 或 **ld（静态连接器命令）**

例如：可用命令“**gcc -o test** main.o test.o”或“**ld -o test** main.o test.o”

生成可执行文件test

这种将一个程序的**所有关联模块**对应的目标代码文件结合在一起，以形成一个**可执行文件**的过程称为**链接**，由专门的连接程序（Linker，也称为**链接器**）来实现。

13015 计算机系统原理【第四章】

考点2 可重定位目标文件和可执行文件不同

例如，通过“objdump -d test.o”命令显示的可重定位文件 test.o 的结果如下。

```
00000000 <add>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp, %ebp
   3:   83 ec 10                sub    $0x10, %esp
```

通过“objdump -d test”命令显示的可执行文件 test 的结果如下。

```
080483d4 <add>:
80483d4:   55                   push   %ebp
80483d5:   89 e5                mov    %esp,%ebp
80483d7:   83 ec 10             sub    $0x10,%esp
```

可重定位文件

1、单个模块生成的

2、代码总是从0开始

可执行文件

1、多个模块组合而成

2、代码在ABI规范规定的虚拟地址空间中产生。

（对于add函数的起始地址为0x80483d4）

13015 计算机系统原理【第四章】

考点3 链接器

名词解释题：链接器

这种将一个程序的**所有关联模块**对应的目标代码文件结合在一起，以形成一个**可执行文件**的过程称为**链接**，由专门的连接程序（Linker，也称为**链接器**）来实现。

链接器在将多个可重定位文件组合成一个可执行文件时，主要完成**符号解析**和**重定位**两个任务

1、符号解析

目的是将每个**符号的引用**与一个确定的**符号定义**建立关联

重点：符号包括全局静态变量名和函数名，而非静态局部变量名则不是符号。

2、重定位（模块化、效率高）

可重定位文件中的代码区和数据区都是从**地址为0开始**的，链接器需要**将不同模块中相同的节合并起来生成一个新的单独的节**，集合ABI规范确定的虚拟地址空间划分来**重新确定位置**。

这种**重新确定代码和数据的地址**并**更新指令中被引用符号地址**的操作称为：**重定位**（**名词解释题**）

13015 计算机系统原理【第四章】

考点4 ELF目标文件格式

通用目标文件格式→COFF：Ststen V UNIX的早期版本

可移植可执行格式→PE：Windows使用的是COFF的一个变种

可执行可链接格式→ELF（**本教材使用**）

a）**链接视图（可重定位目标文件）**

主要由不同的**节**组成，**节头表**，程序头表（可选）

b）**执行视图（可执行目标文件）**

主要由不同的**段**组成，**程序头表**，节头表（可选）

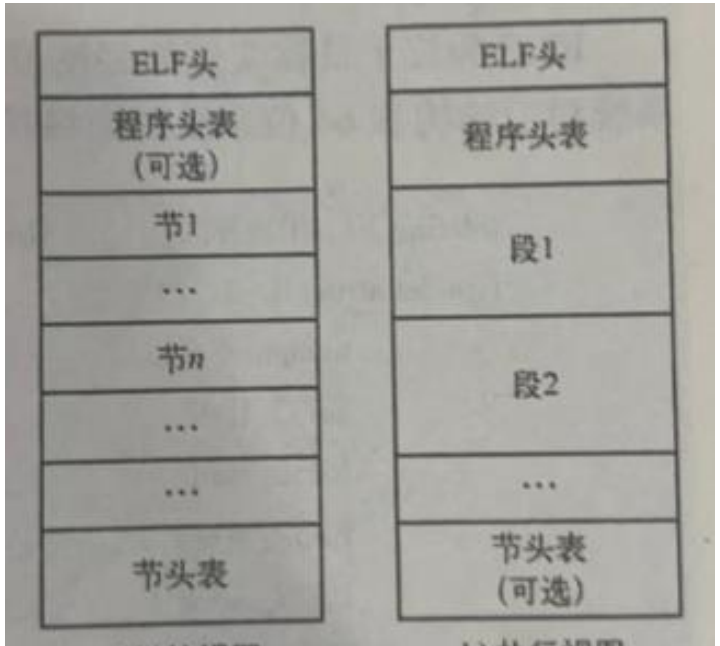


图 4.3 ELF 目标文件的两种视图

13015 计算机系统原理【第四章】

考点4 ELF目标文件格式

a）ELF可重定位目标文件

ELF头：目标文件的起始位置，包含文件结构说明信息

.text节：目标代码

.rodata节：只读数据

.data节：已初始化且初值不为0的全局变量和静态变量，

例如：int x = 10;

.bss节：未初始化或初始化为0的全局变量和静态变量

例如：int y; 或 int z = 0;

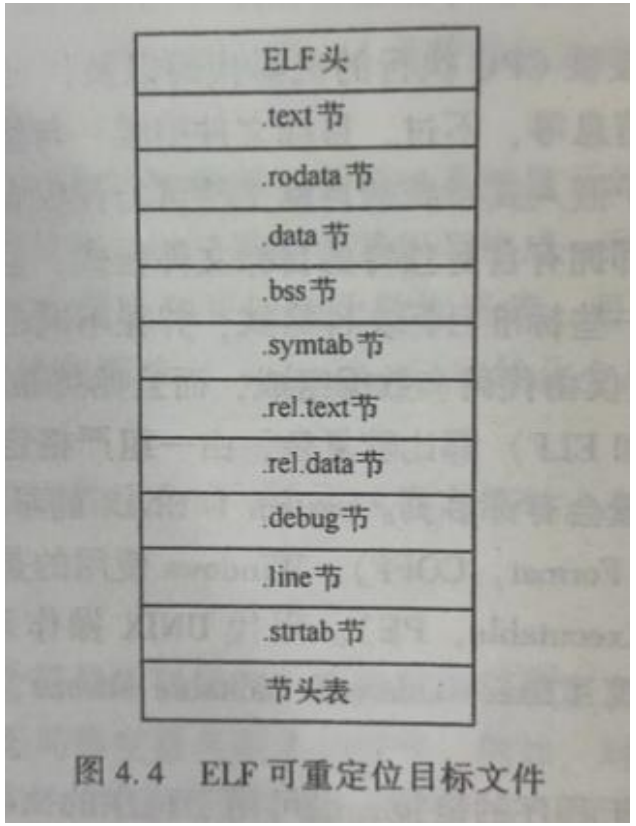


图 4.4 ELF 可重定位目标文件

注意：对于auto型**局部变量**，它们再运行时被**分配在栈中**，因此既不出现在.data节，也不出现在.bss节

13015 计算机系统原理【第四章】

考点4 ELF目标文件格式

a）ELF可重定位目标文件

.symtab节：符号表，在程序中被定义的函数名和全局静态变量名都属于符号，与这些符号相关的信息被保存在符号表中

…

节头表

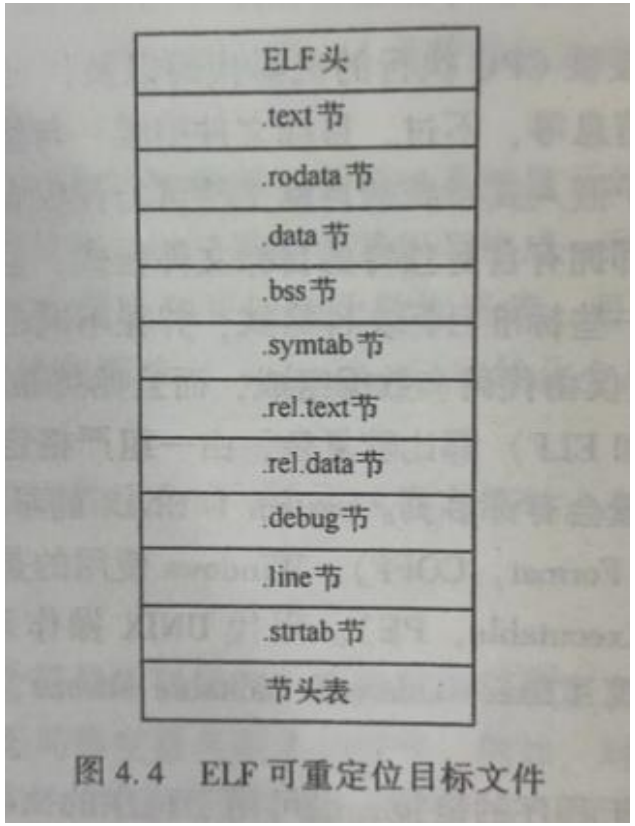


图 4.4 ELF 可重定位目标文件

13015 计算机系统原理【第四章】

考点4 ELF目标文件格式

b）ELF可执行目标文件

程序头表：也称段头表，它是一个结构数组

.init节：用于可执行文件**开始执行时**的初始化工作

当程序开始运行时，系统会在进程**进入主函数main之前**，先执行这个节中的指令代码。

.fini节：**进程终止时**要执行的指令代码

当程序退出时，系统会执行这个节中的指令代码。

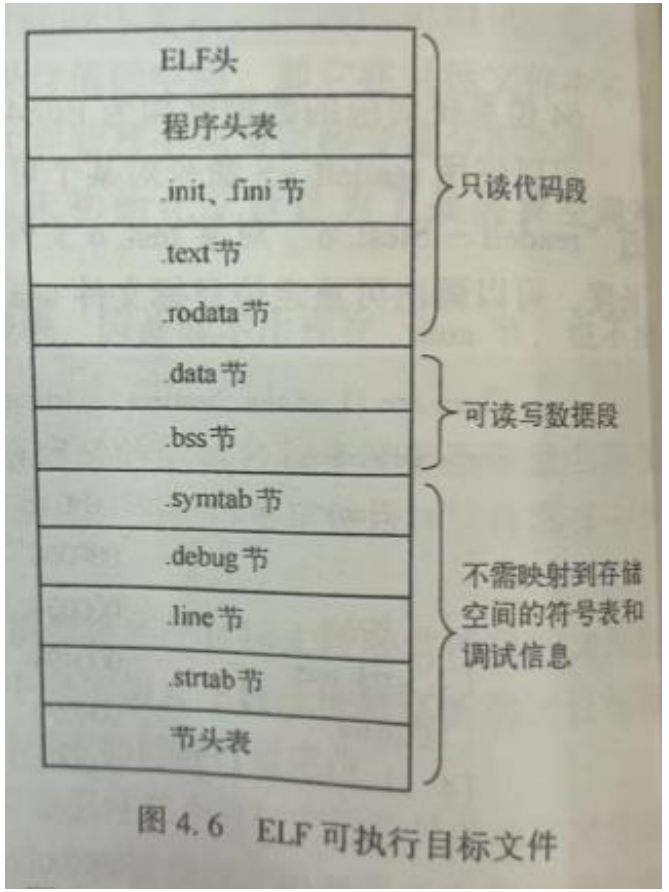


图 4.6 ELF 可执行目标文件

13015 计算机系统原理【第四章】

考点5 堆栈内存

a）运行时堆（堆内存）

低地址→高地址

例如：C语言的malloc()库函数，Java语言的new函数

b）用户栈（栈内存）

高地址→低地址

注意：对于auto型**局部变量**，它们再运行时被**分配在栈中**，因此既不出现在.data节，也不出现在.bss节

13015 计算机系统原理【第四章】

考点6 符号解析与重定位

a）全局符号

包括：**非静态函数名**和**全局变量名**【main.c有buf和main，swap.c有bufp0、bufp1和swap】

b）外部符号（extern）

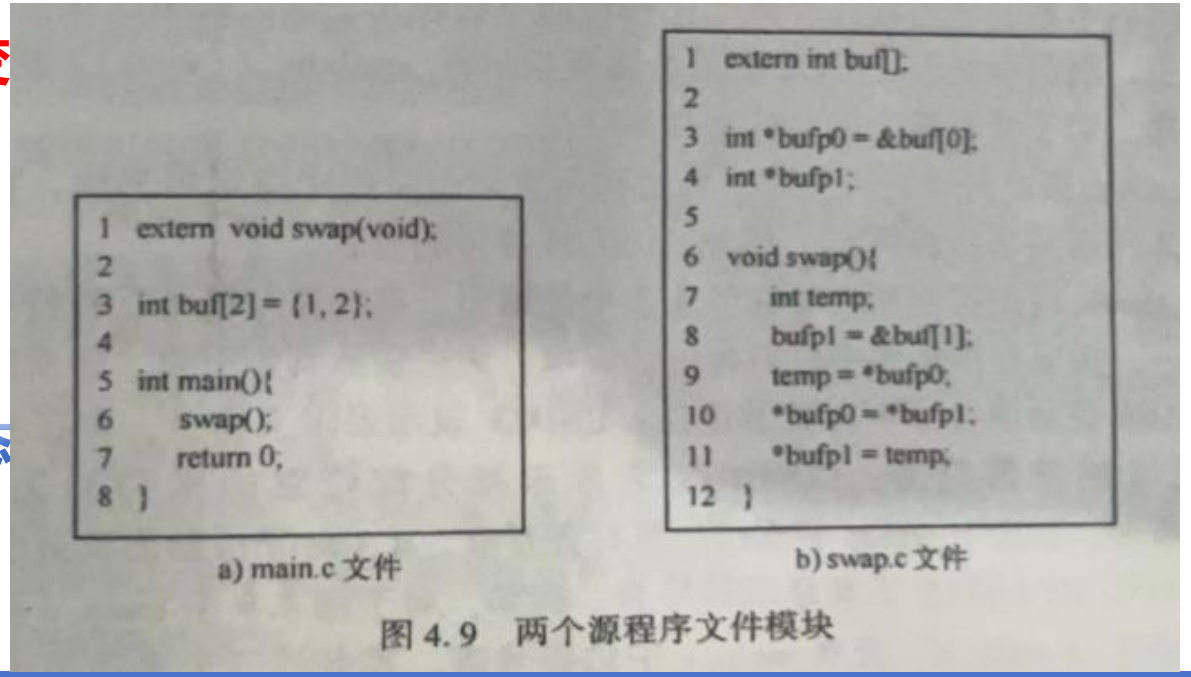
包括：引用在**其他模块**定义的**外部函数名**和**外部变**

c）本地符号

包括：带**static属性**的**函数名**和全局变量名

特别注意：局部变量，不是符号，不在符号表中。

【swap.c中的temp是局部变量，是在运行时动态分配的，因此，它不是符号，不在符号表】



```
1  extern  void swap(void);
2
3  int buf[2] = {1, 2};
4
5  int main(){
6      swap();
7      return 0;
8  }
```

a) main.c 文件

```
1  extern int buf[];
2
3  int *bufp0 = &buf[0];
4  int *bufp1;
5
6  void swap(){
7      int temp;
8      bufp1 = &buf[1];
9      temp = *bufp0;
10     *bufp0 = *bufp1;
11     *bufp1 = temp;
12 }
```

b) swap.c 文件

图 4.9 两个源程序文件模块

13015 计算机系统原理【第四章】

考点7 符号类型、绑定属性和特殊伪节

a）符号类型

未指定→ NOTYP

变量 → OBJECT

函数 → FUNC

节 → SECTION

Type	Bind	Name
OBJECT	GLOBAL	buf
FUNC	GLOBAL	main
NOTYPE	GLOBAL	swap

b）绑定属性

本地 → LOCAL

全局 → GLOBA

弱 → WEAK

Type	Bind	Name
OBJECT	GLOBAL	bufp0
NOTYPE	GLOBAL	buf
FUNC	GLOBAL	swap
OBJECT	GLOBAL	bufp1

```
1  extern  void swap(void);
2
3  int buf[2] = {1, 2};
4
5  int main(){
6      swap();
7      return 0;
8  }
```

a) main.c 文件

```
1   extern int buf[];
2
3   int *bufp0 = &buf[0];
4   int *bufp1;
5
6   void swap(){
7       int temp;
8       bufp1 = &buf[1];
9       temp = *bufp0;
10      *bufp0 = *bufp1;
11      *bufp1 = temp;
12  }
```

b) swap.c 文件

图 4.9 两个源程序文件模块

（通过GCC扩展的属性指示符，**了解**即可）

13015 计算机系统原理【第四章】

考点7 符号类型、绑定属性和特殊伪节

c）特殊伪节

ABS → 表示该符号**不会被重定位**

UNDEF → 表示**未定义符号**

COMMON → 表示**未被分配位置**的**未初始化的变量**，称为COMMON符号

注意：**.bss**和**COMMON**的区别

.bss节：未初始化或初始化为0的全局变量和静态变量

符号冲突处理

.bss：初始化为0的变量是强符号，如果有多处定义，链接时会报错

COMMON：未初始化的全局变量是弱符号，允许多重定义，直到链接时才解决，避免编译阶段错误

13015 计算机系统原理【第四章】

考点8 符号解析规则

符号解析的目的是将每个模块中引用的符号与某个目标模块中的定义符号建立关联。

全局符号

一个全局符号

可能是函数，

或者是.data节中具有特定初始值的全局变量（int a = 1），

或者是.bss节中被初始化为0的全局变量（int b = 0），

或者是说明位COMMON伪节的未初始化全局变量（即COMMON符号）（int c）等等。

为便于说明**全局符号的多重定义**问题，本书将（**函数、.data和.bss**节中的全局变量）统称为：强符号

13015 计算机系统原理【第四章】

考点8 符号解析规则

GCC链接器处理多重定义的同名全局符号的规则是什么？ **(简答题)**

规则1：强符号不能多次定义，否则链接错误。

规则2：若出现一次强符号定义和多次COMMON符号或弱符号定义，则按强符号定义为准。

规则3：若同时出现COMMON符号定义和弱符号定义，则按COMMMON符号定义为准。

规则4：若一个COMMON符号出现多次定义，则以其中占空间最大的一个为准。

规则5：若使用编译选项-fno-common，则不考虑COMMON符号，相当于将COMMON符号作为强符号处理。

13015 计算机系统原理【第四章】

考点8 符号解析规则

上图

符号x 在两个模块中都被定义为强符号规则1

下图

符号y在main.c中是强符号，而在p1.c中是COMMON符号规则2 这两个y是同一个变量，即：main.c的y，最终y=200

符号z在两个模块中都没有初始化，都是COMMON符号，根据规则4的，将占空间较大的符号作为唯一定义符号，即：main.c中的z，最终z=2000

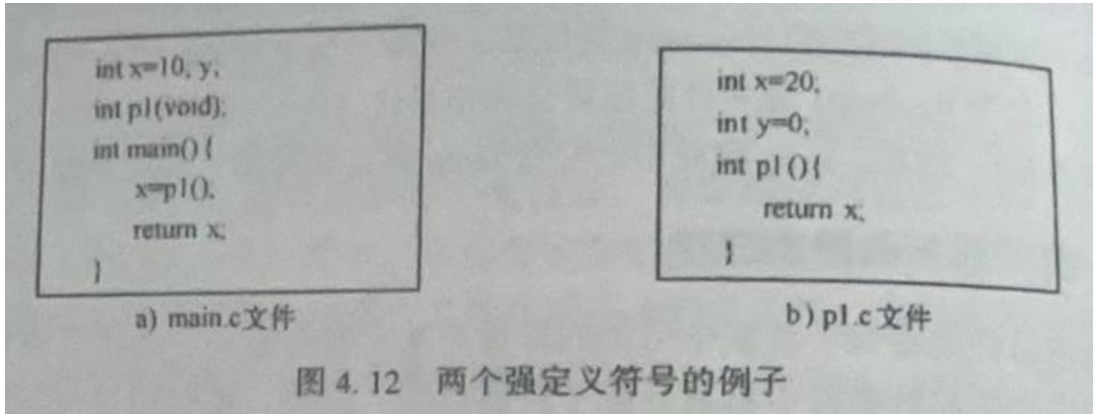


图 4.12 两个强定义符号的例子

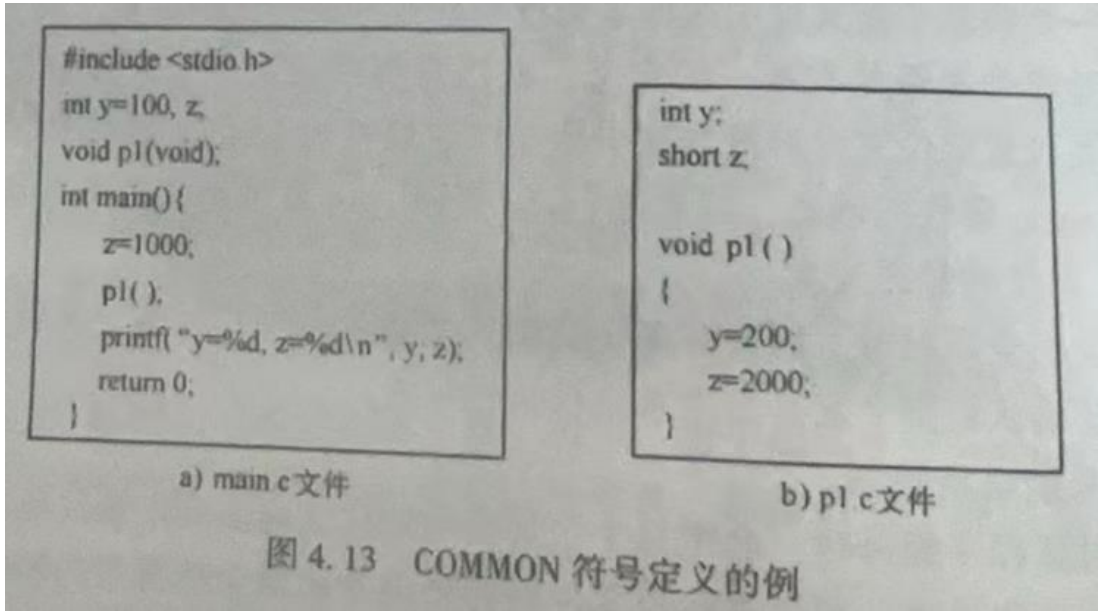


图 4.13 COMMON 符号定义的例

13015 计算机系统原理【第四章】

考点8 符号解析规则

解决同名全局符号引起的问题？ **（简答题）**

1）可以定义为static属性的静态变量

2）尽量要给全局变量赋初值使其变成强符号

3）外部全局变量则尽量使用extern

13015 计算机系统原理【第四章】

考点9 静态链接和动态链接

静态链接

将**多个目标模块**打包成**一个单独的库文件**的机制，这个库文件就是**静态库**

文件扩展名

类UNIX系统中共享库文件扩展名为.a

动态链接

在可执行文件装入或**运行时**被**动态**地装入内存并**自动被链接**，这个过程称为**动态链接**

文件扩展名

类UNIX系统中共享库文件扩展名为.so

Windows系统中共享库文件扩展名为.dll

13015 计算机系统原理【第四章】

考点9 静态链接和动态链接

静态链接和动态链接区别？ （简答题）

链接时机

静态链接：发生在形成可执行程序之前，在编译过程中完成。

动态链接：发生在程序运行时，动态加载所需的模块。

空间资源

静态链接：所有依赖的目标文件都会被包含在最终的可执行文件中，这可能导致多个副本浪费空间。

动态链接：多个程序可以共享内存中的同一库文件，节省资源。

更新升级

静态链接：若静态库更新，所有使用该库的程序都需要重新编译链接，不便于更新。

动态链接：只需替换动态库文件，程序再次运行时会自动加载新版本，简化了更新过程。

13015 计算机系统原理【第四章】

考点10 程序和进程

程序：代码和数据的集合，是静态的。

进程：程序的一次运行过程，是动态的，用正整数表示，简写为**PID**。

注意：一个程序可能对应多个不同的进程。

execve函数：功能是在当前进行的上下文种加载并运行一个新程序。

例如：int execve(char *filename, char *argv[], *envp[])

该函数用来加载并运行可执行目标文件filename

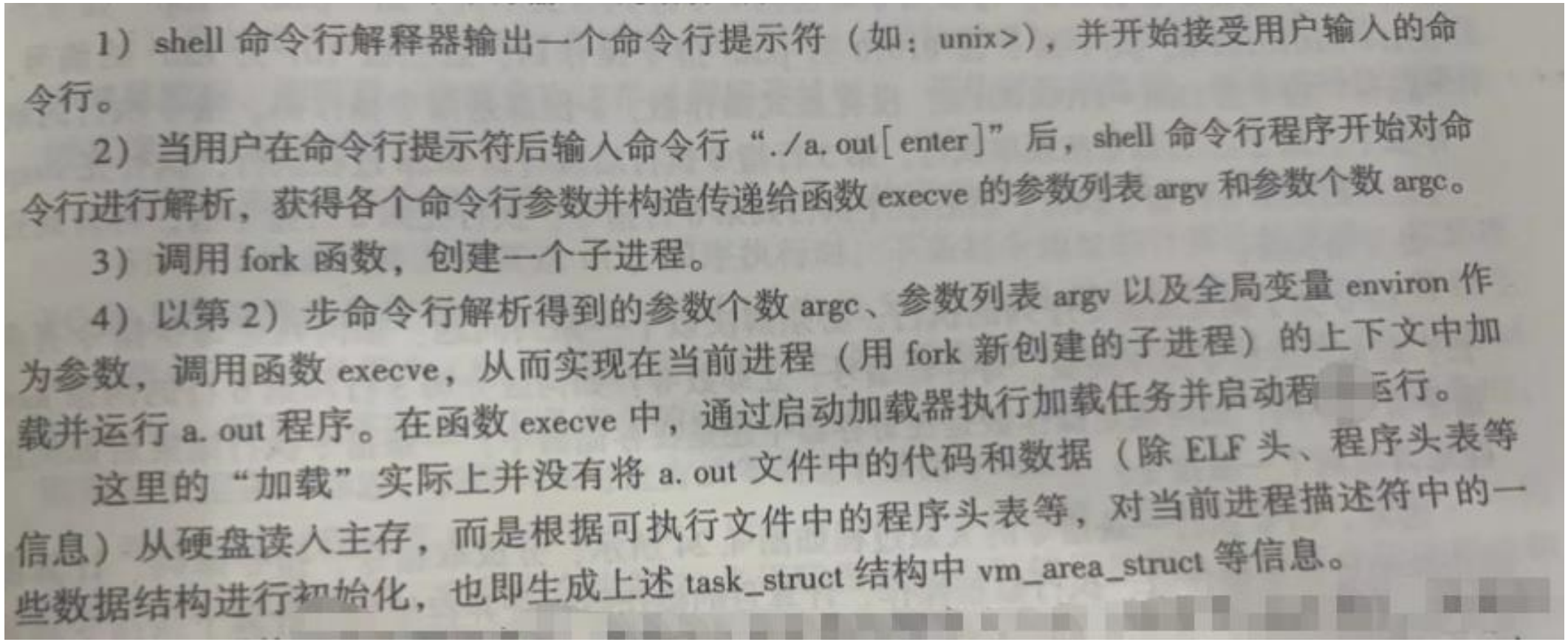
fork函数

在父进程中可通过fork函数创建一个子进程

13015 计算机系统原理【第四章】

考点10 程序和进程

通过shell命令行输入可执行文件名a.out进行程序加载的过程**（简答题）**



1）shell 命令行解释器输出一个命令行提示符（如：unix>），并开始接受用户输入的命令行。

2）当用户在命令行提示符后输入命令行“./a.out[enter]”后，shell 命令行程序开始对命令行进行解析，获得各个命令行参数并构造传递给函数 execve 的参数列表 argv 和参数个数 argc。

3）调用 fork 函数，创建一个子进程。

4）以第 2）步命令行解析得到的参数个数 argc、参数列表 argv 以及全局变量 environ 作为参数，调用函数 execve，从而实现在当前进程（用 fork 新创建的子进程）的上下文中加载并运行 a.out 程序。在函数 execve 中，通过启动加载器执行加载任务并启动程序运行。

这里的“加载”实际上并没有将 a.out 文件中的代码和数据（除 ELF 头、程序头表等信息）从硬盘读入主存，而是根据可执行文件中的程序头表等，对当前进程描述符中的一些数据结构进行初始化，也即生成上述 task_struct 结构中 vm_area_struct 等信息。

13015 计算机系统原理【第四章】

考点11 CPU执行指令及功能

指令周期（名词解释题）

CPU取出并执行一条指令的时间。

CPU执行一条指令的大致过程？（简答题）

1、取指令

2、指令译码

3、计算源操作数地址并取操作数

4、执行数据操作

5、计算目的操作数地址并存结果

6、计算下条指令地址

13015 计算机系统原理【第四章】

考点11 CPU执行指令及功能

指令功能的基本操作？ **（简答题）**

1、读取某存储单元内容，并将其装入某个寄存器。【读内存】

2、把某个寄存器中的数据存储到给定的存储单元中。【写内存】

3、把一个数据从某个寄存器**传送**到另一个寄存器或者ALU的输入端。【传送】

4、在ALU中进行某个算术运算或逻辑运算，并将结果传送到某个寄存器。**【运算写结果】**

13015 计算机系统原理【第四章】

考点12 中断和异常

打断程序正常执行的事件（简答题）

1、非法操作码

2、页故障

3、运算结果溢出或除0错误等

4、CPU收到外部中断请求信号

CPU除了能够正常地不断执行指令以外，还必须具有程序正常执行被打断时的处理机制，这种机制称为异常控制，也称为**中断机制**。

打断程序正在执行的时间分为两大类：

（1）内部异常：结果异常、除0等

（2）外部中断：采样计时时间到、网络数据包到达等

13015 计算机系统原理【第四章】

考点12 中断和异常

CPU对异常和中断的响应过程的步骤（简答题）

1、保护断点和程序状态

通过**程序状态字**保存运行程序的状态信息

2、关中断

通常通过设置**中断使能位来实现**，当置1，开中断，表示允许响应中断，反之，关中断。

3、识别异常和中断事件并转响应处理程序

13015 计算机系统原理【第四章】

考点13 指令流水线

指令的处理过程可以归纳为以下4个阶段（简答题）

1、取指令**并**PC加1（IF）

2、译码**并**读寄存器（ID）

3、运算**或**读存储器（EX）

4、结果写回（WB）

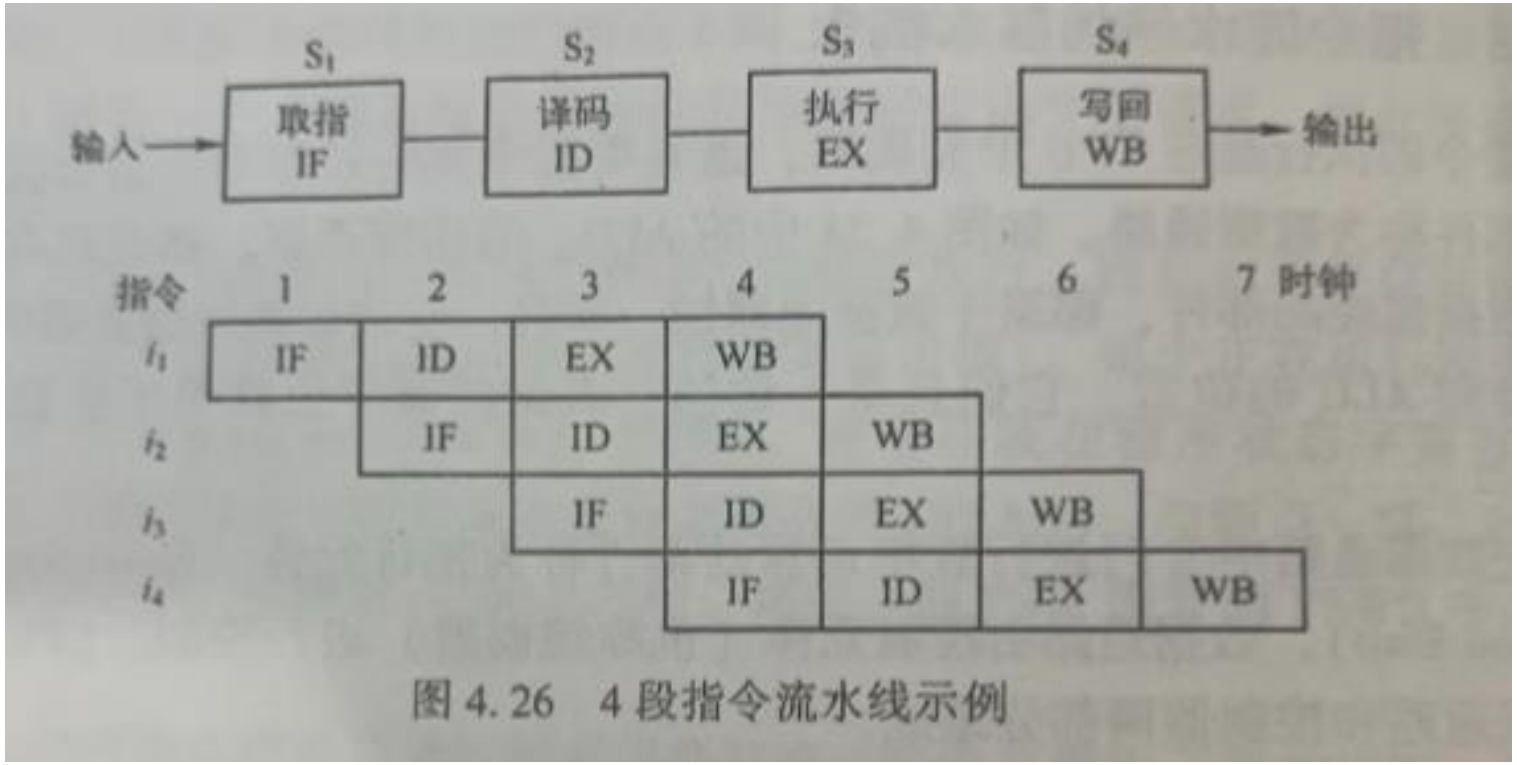


图 4.26 4 段指令流水线示例

指令add → 4个阶段

指令load → 4个阶段

指令mov → IF、ID和WB

指令store → IF、ID和WB

13015 计算机系统原理【第四章】

考点13 指令流水线 ps作为时间单位，代表的是皮秒，即一万亿分之一秒（10^-12秒）。

指令译码—60ps；存储器读或写—200ps；PC加1—40ps；寄存器读或写—50ps；ALU—100ps；

串行 **(助记：AL4 MS3)**

add指令执行时间：**IF+ID+EX+WB = 200+60+100+50=410ps　　【运算ALU】**

load指令执行时间：**IF+ID+EX+WB = 200+60+200+50=510ps　【读存储器】**

mov指令执行时间：**IF+ID+WB = 200+60+50=310ps**

store指令执行时间：**IF+ID+WB = 200+60+200=460ps**

以上加和总时间：1690ps

流水线

若流水段数为M，每个流水段的执行时间为T，则理想状态下，N条指令的执行总时间：

（M-1+N）×T → （4-1+4）×200 = **1400ps**

例如：对于4段流水线，假定某程序有N条指令，理想情况下，总时间为： （3+N）× 200ps

当N很大时，流水线方式比串行执行方式要快很多。

13015 计算机系统原理【第四章】

考点13 指令流水线 ps作为时间单位，代表的是皮秒，即一万亿分之一秒（10^-12秒）。

流水线CPU设计原则

（1）指令流水段个数以**最复杂指令所用的阶段数**为准【最复杂是load指令，所以**M=4**】

（2）流水段执行时间以**最复杂操作所用时间**为准　　【最复杂操作时间是200ps，所以**T=200ps**】

在流水线CPU中，每条指令的执行时间为4×200=800ps，比串行最长的时间510ps还长，

因此流水线方式并不能缩短一条指令的执行时间。但是，对于整个程序而言，流水线方式可以大大增加指令执行的吞吐率。

谢 谢 大 家